# Implementation of Techniques for Enhancing Security of Southbound Infrastructure in SDN

Uday Tupakula, Kallol Krishna Karmakar, Vijay Varadharajan, Ben Collins
Advanced Cyber Security Engineering Research Centre, The University of Newcastle, Newcastle, Australia
[uday.tupakula, kallolkrishna.karmakar, vijay.varadharajan, ben.collins]@newcastle.edu.au

2022 13th International Conference on Network of the Future (NoF) | 978-1-6654-7254-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/NOF55974.2022.9942644

***Abstract*—In this paper we present techniques for enhancing the security of south bound infrastructure in SDN which includes OpenFlow switches and end hosts. In particular, the proposed security techniques have three main goals: (i) validation and secure configuration of flow rules in the OpenFlow switches by trusted SDN controller in the domain; (ii) securing the flows from the end hosts; and (iii) detecting attacks on the switches by malicious entities in the SDN domain. We have implemented the proposed security techniques as an application for ONOS SDN controller. We have also validated our application by detecting various OpenFlow switch specific attacks such as malicious flow rule insertions and modifications in the switches over a mininet emulated network.**

***Index Terms*—Software Defined Networking (SDN) Security, OpenFlow Switch Attack, Security Application, Flow attacks.**

## I. Introduction

One of the main enablers of SDN is the programmable data plane. They empower the SDN Controller with programmable packet processing based on network applications running on the control plane. The data plane hosts soft/virtual or hardware switches which are different from the legacy network routers and switches. They lack intelligence as the SDN Controller controls the routing or packet forwarding decisions. The SDN Controller uses OpenFlow to communicate with the data plane switches, known as OpenFlow switches or devices. These OpenFlow switches consist of only flow rules to route the network packets within the network infrastructure. These OpenFlow Switches are vulnerable to attacks due to the lack of intelligence and close connectivity to the adversary. An adversary connected to these OpenFlow devices can easily tamper/poison the flow rules. In this paper we present SouthBound Security Application (SBSA) to safeguard the data plane elements such as SDN switches and end host communication from attacks.

The paper is organised as follows. In Section II, we present the attacker model. In section III, we first present SBSA and discuss how SBSA helps to enhance the security of the South bound infrastructure. In Section IV, we provide details on the implementation of the SBSA and Section V concludes.

## II. Attacker Model

In this paper we consider attacks on the switches and end host communications in SDN enabled networks. Switches are the critical components that are connected to the Controller in the control plane and end hosts in the data plane. Hence, they are vulnerable to attacks from devices in both planes. For instance, if the attacker is able to access the switch from the devices in the data plane or by using malicious applications in the NBI, s/he can alter the actions that are configured in the existing rules, or insert new rules for performing malicious activities. Often, the end hosts flows are insecure in typical applications such as control systems and IoT as these devices do not have any support for securing the flows due to legacy applications and/or lack of resource. So the attackers can easily intercept and modify the flows between the end hosts during transit. Hence there is need to secure these flows from attacks. Let us discuss some of the entry points for attackers to generate attacks in the data plane in SDN enabled networks.

- In SDN architecture, the OpenFlow switches are loosely attached to the Controller. The Controller passes and receives feedback from the OpenFlow switches using a secure Transport Layer Security (TLS) link. Most of the Controllers disable TLS while communicating with the OpenFlow switches [1]. Thus, it creates a loophole for the adversaries to push malicious instructions into the switch. Even for the cases where SSL/TLS secure communication is enabled in SDN, vulnerabilities such as Heartbleed provide an opportunity to exploit the weakness in these protocols to generate attacks on the switches.
- The controller often runs a number of applications, each of which includes specific policies and enforces the corresponding rules to be deployed. Often multiple applications can initiate commands to simultaneously manage the same physical network. In many cases, these applications can be designed by third parties that can be buggy or malicious. The current Controllers do not validate the commands initiated by the applications [2]. Hence it is easy for an attacker to make use of malicious applications to generate attacks on the switches.
- Since the switches are directly connected to end hosts, the attackers can make use of end hosts to inject malicious flow rules or attackers can exploit a vulnerability in the switch to obtain unauthorised access and alter the existing rules by perhaps changing the deny actions to permit, changing permit actions to deny or providing priority to the flows generated by attacker. Such attacks are easily possible in cloud environments [3] where SDN is used for the management of tenant virtual machines. For instance, in case of public IaaS clouds the tenants have complete control on the operating systems and applications running in their virtual machines. Hence the malicious tenants can run tools such as Kali Linux in their virual machines to exploit vulnerabilities such as VM escape to access the

privileged domain and perform any malicious activity on the SDN switches.

## III. Our Approach

Our work makes use of SBSA for enhancing the security of south bound infrastructure in SDN enabled networks. The SBSA is designed and developed to make use of existing components in the SDN controller and we have also developed different software modules and specific functions for achieving three main goals. First, it ensures that the commands from the SDN Controller and applications to the switches are validated and flow rules are securely installed in the OpenFlow switches. Secondly, to dynamically provide confidentiality and integrity for insecure flows between the end hosts within the SDN Domain. Third goal is to detect attacks on the SDN switches by malicious entities such as end hosts, northbound applications and attacks with spoofed identity of the controller.
In this paper, we consider that the SDN Controller is trusted within its domain and our SBSA is hosted on the trusted controller. We also assume that the Controller and switches has a pair of public and private keys and store them. Now let us discuss how our security application achieves these goals.

### *A. Secure Flow Rule Installation*

In typical network environments and emerging 5G/6G networks, the configuration commands to the switches are issued by different applications and Controller in multiple domains. Also, the controllers in each domain do not validate the configuration commands from different application in the North Bound Interface.The SBSA is used for validating the flow rule configuration commands from different applications and controller and securely installing the flow rules in the switches. A detail discussion on the security application architecture for validating the configuration commands for switches is presented in [4].
The Controller uses OpenFlow to send instructions to the OpenFlow switches. OpenFlow uses TLS to secure the OpenFlow communication [5], [6], [1]. Due to performance issues, TLS is disabled by most of the Controllers available in the market [1]. We have extended OpenFlow by properly utilizing TLS. In our case, TLS helps to securely install the flow rules into OpenFlow switches. In this section, we present, the TLS extension and explain how it is used to securely install flow rules into the OpenFlow switches. It has five major phases. We have illustrated the protocol in Figure 1 and Table I.
**Phase1: Setup Security Capabilities**
In this phase, both the Controller and switch agrees on the cipher suite they will be using during the handshake process. At first switch issues a *Client Hello* message, which contains a nonce (a random number), available cipher suits, session ID, and available compression method. In response, Controller sends a *Server Hello* message containing, a server nonce (replays the switch selected nonce), previous session ID, selected cipher suite and compression method for the handshake process. It is represented as STEP 1 in Table I.
**Phase2: Server Authentication and Key Exchange**
In this phase, Controller generates it's public/private keys and

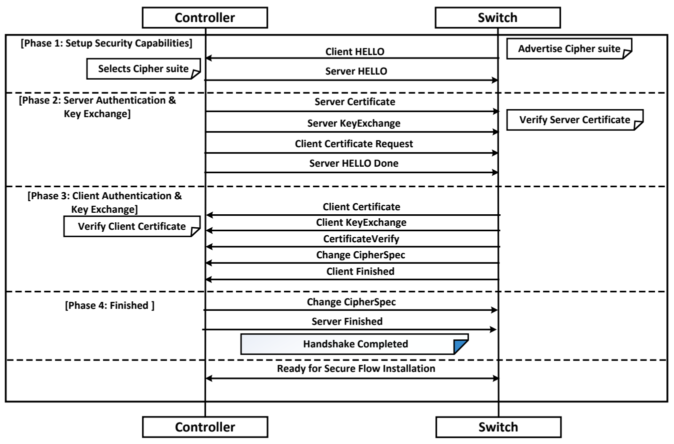


Fig. 1: TLS for secure flow installation

responds to the OpenFlow switch by sending its certificate. The Controller creates a hash of the information and signs its own private key. The Controller certificate contains the Controller public key. Hence, the switch receives the public key of the Controller. The OpenFlow switch verifies the Controller certificate. The exchange happens during the key exchange phase. The type of key and size depends on the previously negotiated encryption method. The Controller also issues a request to send the OpenFlow switch certificate. The message contains a new nonce generated by the Controller. Finally, the Controller finishes the phase by sending a *Server Hello Done* message. It is represented as STEP 2 and 3 in Table I.
**Phase3: Switch Authentication and Key Exchange**
Verification of switch certificate is optional, and, in most cases, it was overlooked while implementing TLS in OpenFlow [7]. In this phase, switch responds with its certificate to the Controller. The OpenFlow switch also sends *certificate verify* message to ensure that the switch is legitimate and has its private key. This message is signed by the private key of the switch. It is represented as STEP 4 in Table I.
The Controller generates a pre-master secret and encrypts it with the switches public key. Finally, it creates a hash of it and signs it with its private key. It is represented as STEP 5 in Table I.
**Phase 4: Finished**
Both parties use the pre-master secret and nonces to calculate the master key (a symmetric key). It is represented as STEP 6 in Table I.
**Phase 5: Secure Flow Installation**
Switch acknowledges the Controller about the reception of the pre-master secret and the Controller responds by acknowledging the reception. It is represented as STEP 7 and 8 in Table I. Thus, the Controller uses the master key to install the flow rules securely.
**Notations:**

- Version information of switch and Controller is represented as $V_{SW_i}$ and $V_C$ respectively.
- Cipher suite advertisement by the switch is presented as $Suite_{SW_i}$.
- Controller has a pair of public and private key

$(K_{PU(C)}/K_{PR(C)})$.
- Cipher suite selected by the Controller is $Suite_C$.
- Nonce generated by the switch $n_{SW_i}$.
- Nonce generated by the switch $n_C, n'_C$.
- $KM_{C-SW_i}$ is the master key for any particular Controller $(C)$ to switch $(SW_i)$ communication.
- $P_{MS}$ is pre-master secret.
- Controllers certificate is represented by $Cert_C$.
- Switches certificate is represented by $Cert_{SW_i}$.
- Hash function is represented as $h(.)$.
- A function to calculate the master key is represented as $f(.)$, which is a combination of MD5 and SHA.
- A certificate request is represented as $req$.
- An acknowledgement is represented as $ack$.

TABLE I: Modified TLS for Secure flow installation

**STEP 1:** $SW_i \to C :< SW_i, V_{SW_i}, Suite_{SW_i}, n_{SW_i} >$
**STEP 2:** $C \to SW_i :< V_C, Suite_C, n_{SW_i}, Cert_C > [h(V_C, Suite_C, n_{SW_i})]_{K_{PR(C)}}$
**STEP 3:** $C \to SW_i :< SW_i, n_C, req > [h(SW_i, n_C, req)]_{K_{PR(C)}}$
**STEP 4:** $SW_i \to C :< C, n_C, Cert_{SW_i} > [h(C, n_C)]_{K_{PR(SW_i)}}$
**STEP 5:** $C \to SW_i :< SW_i, n'_C, [P_{MS}]_{K_{PU(SW_i)}} > [h(SW_i, n'_C, [P_{MS}]_{K_{PU(SW_i)}})]_{K_{PR(C)}}$
**STEP 6:** Both parties calculate $f(n_{SW_i}, n_C, n'_C, P_{MS})$ to get $KM_{C-SW_i}$
**STEP 7:** $SW_i \to C :< C, n'_C, ack > [h(C, n'_C, ack)]_{KM_{C-SW_i}}$
**STEP 8:** $C \to SW_i :< SW_i, n'_C + 1, ack' > [h(SW_i, n'_C + 1, ack')]_{KM_{C-SW_i}}$

## *B. Securing the Flow Information*

As already mentioned, the end hosts flows are insecure in typical applications such as control systems and IoT. So there is need to secure the flows during the transit. In the current SDN network by default, none of the Controller's provides on-demand confidentiality service for user flows. Our SBSA provides confidentiality service to the users or devices if required. The administrator can specify the end hosts (with limited resources) that require secure communication. We have developed a policy specification language using JSON for the specification of secure communication policies in SDN domain. A detail discussion on the policy specification language can be found in [4]. Now, we explain the mechanism in detail in this section.

Let us consider a scenario shown in Figure 2, where a user $U1$ connected with switch $SW1$ is trying to communicate with user $U2$ in switch $SW2$. In normal SDN operation, $U1$ sends the packet to $SW1$. Since it is a new communication it will not have any flow rules and $SW1$ sends a *packet_IN* request to the Controller. The Controller knows the network topology and hence deploys the flow rules in the OpenFlow switches that will establish the communication between $U1$ and $U2$. However, the payload for all flows in such communications in SDN is unencrypted.

In SBSA, we are able to provide confidentiality service to the communicating entities, in this case, the hosts. We have policy expressions that specify which of the communication entities would be provided with confidentiality services. The Controller generates symmetric keys for each of these communications according to the policy expressions. The key convention used for the policy expression is $K_{S(PolicyExpressionID)}$. The switches use these symmetric keys to encrypt the flow payload during communication. The agents in the switch help in the encryption and decryption process of the payload. The distribution or sharing the symmetric key with the OpenFlow switches present within a path other than the source and destination switch is entirely dependent upon the Controller and the policy expressions present in SBSA. In exceptional cases, the intermediary OpenFlow switches can request for these symmetric keys for any particular flow. We have introduced new messages in the OpenFlow protocol for this operation, which is shown in Listing 1.

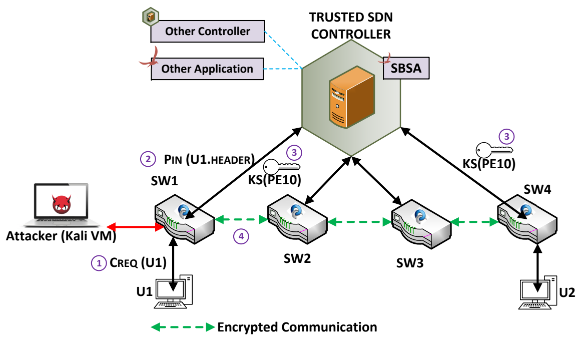


Fig. 2: SouthBound Security Application

In our scenario, we have a policy expression $(PE10)$ which expresses that the communication between $U1$ and $U2$ should be provided with confidentiality service. When the communication request $(C_{REQ(U1)})$ comes to the $SW1$ from $U1$, it issues a *packet_IN* request $(P_{in(U1.header)})$. The SBSA checks the policy and finds the confidentiality service need to be provided for the communicating parties. Hence, it generates the key $K_{S(PE10)}$, deploys the flow rules to $SW1 - SW4$ and also sends the key to them. As shown in Figure 2, $SW1$ uses $K_{S(PE10)}$ to encrypt the payload and send it to the destination switch $SW4$, where it is decrypted and transferred to $U2$. The Controller will not share the encryption key with the intermediary route switches unless authorised in the policy expression or any intermediary route switches requests for the key.

Listing 1: Request message for flow specific symmetric keys
```
/* Request message (Symmetric-key). */
struct ofp_ks_reqmsg {
struct ofp_header header; /* Type
    OFPT_EXPERIMENTER. */
uint32_t ksreqmsg;      /* Symmetric-key
    request
                         /* Reason of request.
                             */
uint8_t ksreqmsg_reason[0];
};
enum onf_ksreqmsg{
REQUEST =1<<0 /* 1 indicates that it's
    requesting for Symmetric-key */
ADDITIONAL INFO=1<<0 /* 1 indicates that the
    request message contains
some reason. */
}
```

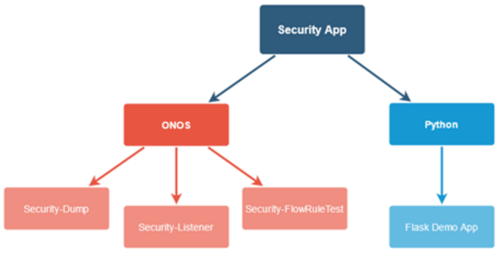


Fig. 3: Attack Detection Components

### *C. Detecting attacks on Switches*

Now let us discuss how SBSA can be used to detect attacks on the switch. The SBSA logs all the configuration commands from the trusted controller to the switches. At regular intervals, the SBSA queries the switches to report the current flow rules in the flow tables. When SBSA receives the flow rules report from each switch, it uses the logs to determine if there are any malicious flow rule insertions, flow rule modifications or flow rule deletions in the switch. If there are any variations in the switch rules compared to the SBSA logs, then the switch is considered to be under attack and an alert is raised to the administrator.

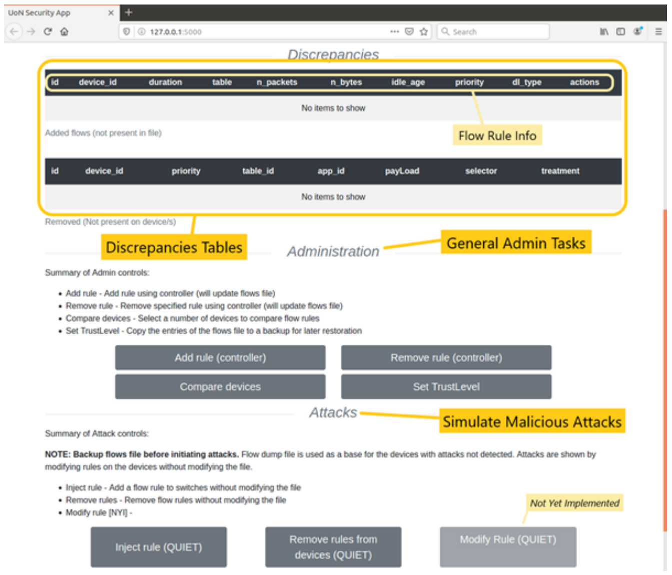
In this section we will discuss the specific components that are developed to detect attacks on the switches. As shown in Figure 3, we have developed three specific functions including security listener, switch dump and security-FlowRuleTest for detecting attacks on the switches. The security listener is used for maintaining a secure log of all the flow rule configuration commands from the trusted controller to the switches. The switch dump is used for determining the flow rules that are currently enforced in the switches. The security-FlowRuleTest is used for detecting the attacks by validating the switch dump report with the security listener report. We have also developed a GUI for demonstration of the application. Furthermore, we have developed a trust model to determine the trust levels of the switches. However, the trust model is not discussed in the current work. We are planning to publish the trust model as a separate work.

## IV. Implementation

We have developed a working prototype of the SBSA presented above. As shown in the Figure 2, we have used Kali Linux virtual machine to maliciously inject new flow rules, modify existing flow rules in the switch and use SBSA to detect the attacks.

Hardware and network setup: We have used the Cybersecurity lab's (Advanced Cyber Security Engineering Research Centre) VMware NSX cluster for implementing the prototype. We have used a virtual machine with 32GB of RAM and 4vCPU (@3.2GHz) for this. The SBSA is hosted on ONOS SDN Controller for controlling the switching devices. Here, mininet is used for simulating the switching devices.

The developed application framework is divided into two sections. The front-end and the back-end. The back-end is the SBSA Application running in ONOS Northbound and the API link with the OpenFlow Switch services. The front-end of the application is developed using python-Flask, a lightweight web framework. The web application is completely isolated



| id | device_id | duration | table | n_packets | n_bytes | idle_age | priority | dl_type | actions |
|---|---|---|---|---|---|---|---|---|---|
| No items to show | | | | | | | | | |

Added flows (not present in file)

| id | device_id | priority | table_id | app_id | payLoad | selector | treatment |
|---|---|---|---|---|---|---|---|
| No items to show | | | | | | | |

Removed (Not present on device/s)

Summary of Admin controls:

- Add rule - Add rule using controller (will update flows file)
- Remove rule - Remove specified rule using controller (will update flows file)
- Compare devices - Select a number of devices to compare flow rules
- Set TrustLevel - Copy the entries of the flows file to a backup for later restoration

Summary of Attack controls:

**NOTE: Backup flows file before initiating attacks.** Flow dump file is used as a base for the devices with attacks not detected. Attacks are shown by modifying rules on the devices without modifying the file.

- Inject rule - Add a flow rule to switches without modifying the file
- Remove rules - Remove flow rules without modifying the file
- Modify rule [NYI] -

Fig. 4: Front-end of the SBSA application

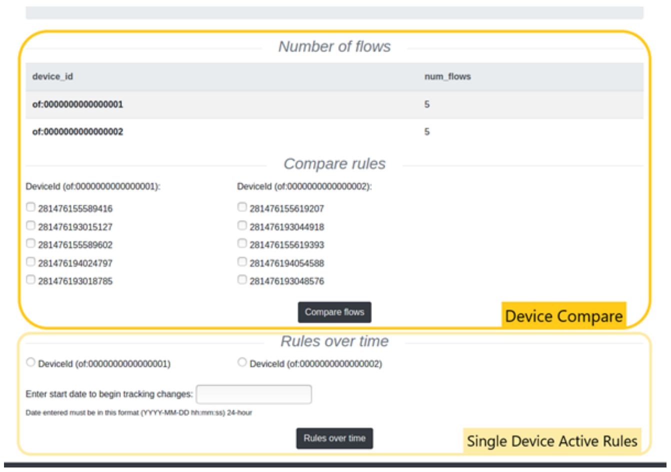
from the ONOS implementation. This front end helps us in the administration of the SBSA and demonstrating switch related attacks. Although ONOS provides its own Web UI for establishing UI content to run alongside ONOS applications, we found it to be limited to the interface we wanted to create to demonstrate the features of SBSA application. The ONOS Web UI is a hybrid of the Angular typescript application framework and its custom framework.

We are using ONOS Restful API to establish communication between the front end and the SBSA application running in ONOS northbound API. This front-end application is also capable of interacting with the OpenFlow switch CLI. We have included this feature for two reasons. First, this will allow us to directly interact with the OpenFlow switches and help in measuring their trust and security status. Secondly, during the attacks demonstration phase, this will help us to inject flow rules directly into the OpenFlow switches. Figure 4 shows the front-end interface of the application. Now we will demonstrate different sections of the front-end.

The Front-end has two control panels, one for SBSA application control (also known as Admin Control Panel) and the other panel holds the controls for simulation of the attack scenarios (Attack Scenarios).

**Admin Control Panel:** The Admin Control Panel implements several general administration controls for SBSA, such as legitimate modifications to rules and comparing the installed flow rules existing on various OpenFlow devices. With the help of this interface and SBSA, an administration can add rules, remove rules, compare devices and set TrustLevels (TL) for devices.

- **Modification:** We are using JSON strings to present, add and modify the flow rules. The JSON string holds the device ids specified by the user, and the application makes post requests to the ONOS flows API to add the defined flow rule to the devices. This interface also allows the administrator to remove the existing rules. The list of rules to remove is used to iterate delete requests with the



Number of flows

| device_id | num_flows |
|---|---|
| of:0000000000000001 | 5 |
| of:0000000000000002 | 5 |

Compare rules

| DeviceId (of:0000000000000001): | DeviceId (of:0000000000000002): |
|---|---|
| ☐ 281476155589416 | ☐ 281476155619207 |
| ☐ 281476193015127 | ☐ 281476193044918 |
| ☐ 281476155589602 | ☐ 281476155619393 |
| ☐ 281476194024797 | ☐ 281476194054588 |
| ☐ 281476193018785 | ☐ 281476193048576 |

Rules over time

○ DeviceId (of:0000000000000001) ○ DeviceId (of:0000000000000002)

Enter start date to begin tracking changes:

Date entered must be in this format (YYYY-MM-DD hh:mm:ss) 24-hour

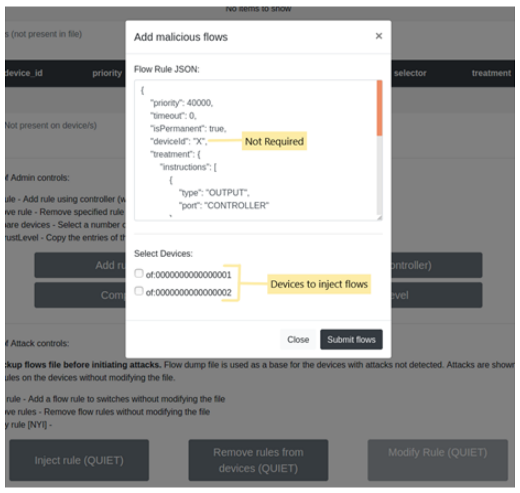
Fig. 5: Flow rule comparison site

ONOS flows API. The removed flows are added to the log file with the "REMOVED" tag and the log also holds its timestamp.

- **Comparison:** The administrator or the user can use this interface to compare flow rules already/previously installed in the devices. The administrator can select multiple devices for a domain-wide device status comparison. The front-end also has a separate comparison portal (Figure 5). Figure 5 shows a table comparing the number of rules in each of the selected switches. The flow rule data is retrieved from the ONOS flows API.
The administrator can also compare the active rules in switches over a selected period. Here, they will choose the start date and time, and the application returns the flow rule changes recorded in the log file for that specific device in that period. Here, the flow rules are tagged as "ADDED" and "REMOVED" depending on the flow rules' status. On the back end, the web application reads the log entries and converts them to JSON objects for ease of comparison. Later, these JSON objects are compared by the flask web application.
- Trust Level: In this current implementation, we are using static trust level. We have already extended our application to assign trust levels dynamically. In future, we plan to publish that as a separate work. For this work, the statically assigned TrustLevels are saved to a JSON file under "config/TrustLevel.json" and are overwritten when the Administrator specifies a different trust level for devices. The TrustLevel parameter is vital for routing confidential data-packets.

**Attack Controls** These controls are specific to the simulation

- **Injection of Flow Rules:** Using this control panel malicious flow rules can be injected into any OpenFlow switch. However, unlike the legitimate case of flow inclusion, these flows are not being checked, or the flow attributes are fake. The fake flow rules are specified as JSON objects (as shown in Figure 6). The JSON object can be modified to include either a legitimate device id



Add malicious flows

Flow Rule JSON:

```
{
    "priority": 40000,
    "timeout": 0,
    "isPermanent": true,
    "deviceId": "X",
    "treatment": {
        "instructions": [
            {
                "type": "OUTPUT",
                "port": "CONTROLLER"
```

Select Devices:

☐ of:0000000000000001
☐ of:0000000000000002

**kup flows file before initiating attacks.** Flow dump file is used as a base for the devices with attacks not detected. Attacks are show
ules on the devices without modifying the file.

rule - Add a flow rule to switches without modifying the file
ve rules - Remove flow rules without modifying the file
y rule [NYI] -

Fig. 6: Injecting fake flow rules as JSON object.

of attack scenarios. The controls will help us simulate attacks on the OpenFlow switches, such as injecting malicious flow rules to the OpenFlow switches.

or a fake device id specified by any adversary. Similarly, other attributes of the flow rules can be forged. In some cases, they could be used to route the traffic maliciously. The SBSA logs all flow rule operations to the log file.

- **Deletion of Flow Rules:** This panel also allows the adversary to modify existing flow rules and remove flow rules from any OpenFlow switches maliciously. The SBSA application logs both removal/modification incidents as well in its log file. These logs help SBSA to detect and report incidents to the administrator.

## V. Conclusion

In this work, we have presented techniques for enhancing security of the south bound infrastructure in SDN. In particular, we have developed SBSA to achieve the following goals: (i) validation and secure configuration of flow rules in the OpenFlow switches by trusted SDN controller in the domain; (ii) securing the flows from the end hosts; and (iii) detecting attacks on the switches by malicious entities in the SDN domain.

## References

[1] K. Benton, L. J. Camp, and C. Small, "Openflow vulnerability assessment," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 151–152.
[2] C. Yoon et al., "Flow wars: Systemizing the attack surface and defenses in software-defined networks," *IEEE/ACM TON*, vol. 25, no. 6, 2017.
[3] K. Thimmaraju *et al.*, "Taking control of sdn-based cloud systems via the data plane," in *Symp. on SDN Research*. ACM, 2018, p. 1.
[4] K. Karmakar, "Techniques for securing software defined networks and services," Ph.D. dissertation, University of Newcastle Australia, 2019.
[5] T. Dierks and E. Rescorla, "The transport layer security (tls) protocol version 1.2," Tech. Rep., 2008.
[6] N. McKeown et al., "Openflow: enabling innovation in campus networks," *ACM SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, 2008.
[7] B. Agborubere et al. and E. Sanchez-Velazquez, "Openflow communications and tls security in software-defined networks," in *Internet of Things 2017 IEEE International Conference on*. IEEE, 2017, pp. 560–566.